Lab 2: Cryptography

Nathan Rayburn

1 Mode of Operation

1.1 Introduction

Encryption

In our current implementation, **t** functions as the key stream for the chain. The problem with this approach is that **t** is XORed with the IV after it has been generated, not before. This makes the IV completely useless. This design can be compromised if we encounter two separate messages that begin with the same 16-byte block. If this condition is met, the key stream for each subsequent iteration used to cipher the rest of the message can be decrypted. To retrieve **t**, one can simply XOR $c[1]$ with $c[0]$, effectively eliminating the IV from the equation.

1.2 Cracking

c_1 and c_2 are the ciphertexts, IV_1 and IV_2 are the initialization vectors. If t_1 equals t_2, it means that we can decrypt the whole ciphered c_2. We also need m_1 which is the initial plaintext message for c_1.

$$t_1 = c_1[1] \oplus IV_1$$
$$t_2 = c_2[1] \oplus IV_2$$

If $t_1 = t_2$, therefore:

$$\text{current\_stream} = m1\_blocks[i] \oplus c1\_blocks[i]$$

Key stream t_1 can decrypt c_2 cipher and vice versa, now we can retrieve the plaintext.

$$pt = \text{current\_stream} \oplus c2\_blocks[i]$$

The decrypted blocks are then concatenated together to form the decrypted message. Finally, we join each block and capture the flag.

```
'This is a long enough message for it to be secure. The secret flag
    is laurels\x03\x03\x03'
```

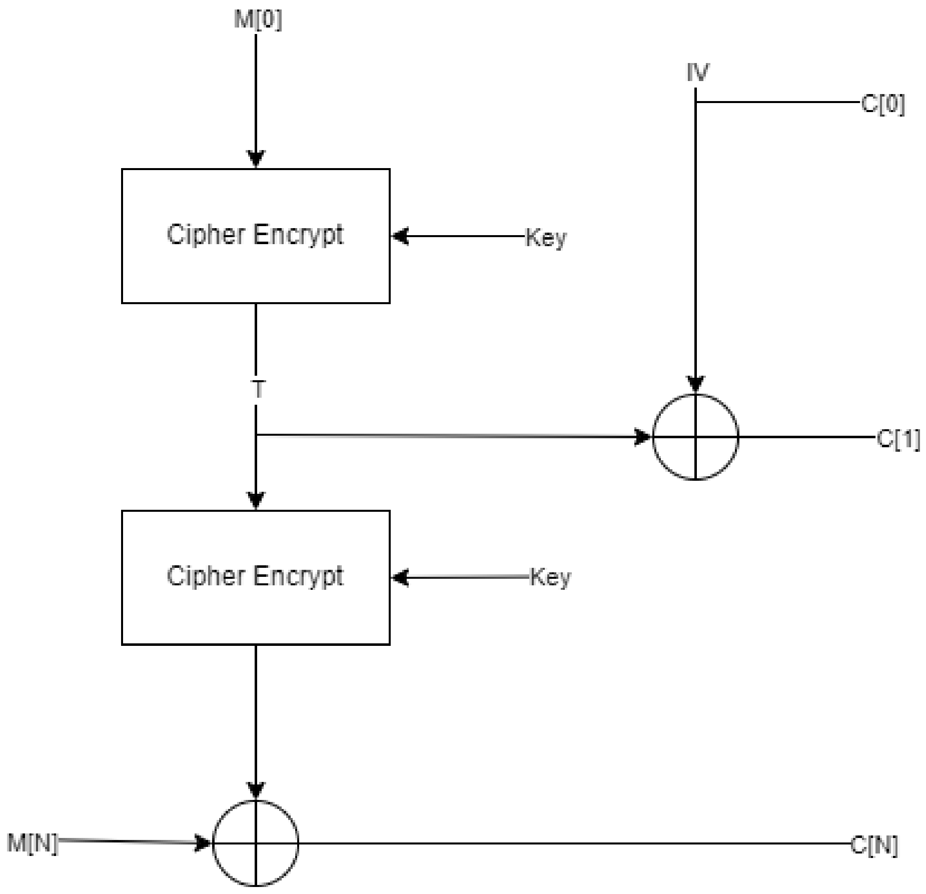

Figure 1: Encryption

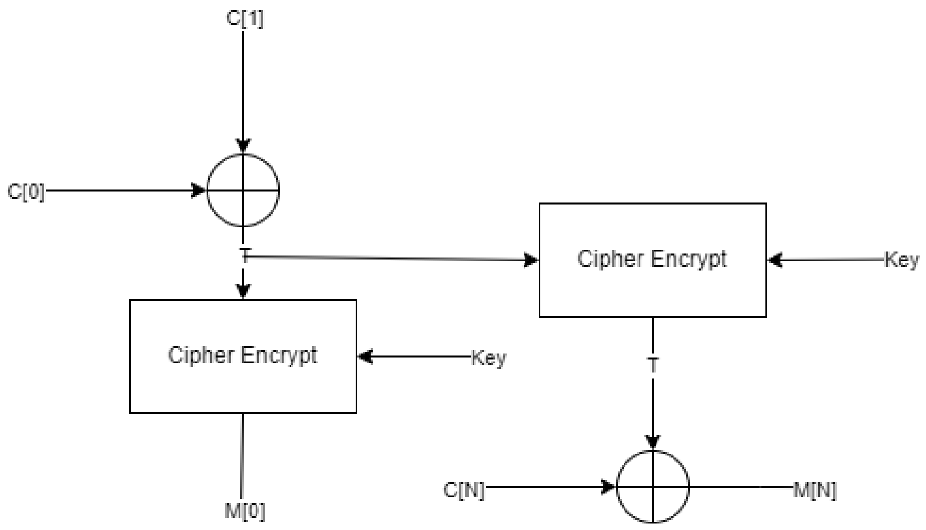

Figure 2: Decryption

2 Enc And Mac

2.1 Introduction

The primary security flaw in this system's implementation lies in the malfunctioning counter. The counter is initialized within a loop, causing it to reset after each iteration, thereby rendering the counter mode (CTR) ineffective. This compromises the security by negating the benefits of the keystream's uniqueness in each encryption block.

Additionally, the keystream is encrypted with 16 bytes of zeros, leading to each 16-byte block lacking any randomness. This significantly undermines the security of the system by compromising the unpredictability which is essential for robust encryption.

2.2 The Math

By applying mathematical formulas, we can determine **V**, a constant utilized in both texts, which will be crucial for later decrypting our plaintext message. Additionally, we need to identify another unknown variable, sigma. Sigma is the keystream of the implementation.

We can determine sigma by subtracting our plaintext block from the corresponding ciphered text block within the same message. We can choose any block—first, second, third, etc.—since they all share the same keystream, owing to the flaw of the implementation of the counter mode (CTR).

$$\sigma = ((c1\_blocks[0]) - (m1\_blocks[0])) \mod p$$

The second step is to isolate **V** like so and plug in the sigma we have found previously.

Sigma formula:

$$\sigma = \left(\text{tag1} - \sum_{i=0}^{n} m_i \cdot v\right) \mod p$$

V Isolated:

$$v = \left(((\text{tag1}) - \sigma) \times \text{mod\_inverse}(\sum_{i=0}^{n} m_i, p)\right) \mod p$$

The next step is to find sigma for the second message.

We can take our MAC formula and isolate sigma.

$$\text{MAC} = \sum_{i=0}^{n} m_i \cdot v + \sigma \mod p$$

Here is sigma isolated:

$$\sigma = \left(\text{MAC} - \sum_{i=0}^{n} m_i \cdot v\right) \mod p$$

The issue is that we don't have $m[i]$, we must find an equivalent for it. We can do so by using a formula that we have used previously.

We can find an equivalent by using our previous formula to find sigma:

$$\sigma = (c1\_blocks[0] - m1\_blocks[0]) \mod p$$

By isolating our message we get this:

$$m1\_blocks[0] = (c1\_blocks[0] - \sigma) \mod p$$

So this is for a single block. Now we can generalize it for all blocks of our message $m[i]$:

$$\sum_{i=0}^{n} m_i = \left(\sum_{i=0}^{n} c_i - \sigma\right) \mod p$$

$$=> \sum_{i=0}^{n} m_i = \left(\sum_{i=0}^{n} c_i\right) - n \cdot \sigma \mod p$$

Isolating the sigma from the sum will be useful later since it will be totally isolated in the next equation.

We can finally replace the sum of $m[i]$ in our initial formula and isolate sigma. No more unknown variables.

Initial formula of sigma isolated:

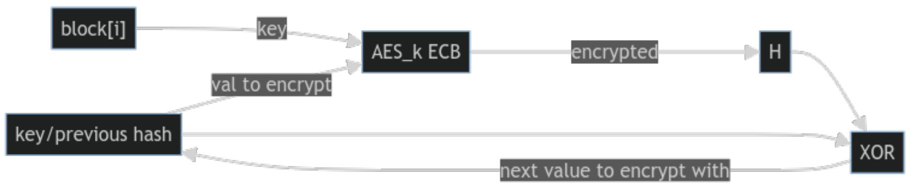

Figure 3: Mac forgery process

$$\sigma = \left(\text{MAC} - \sum_{i=0}^{n} m_i \cdot v \right) \mod p$$

Sum of $c2$ blocks:

$$\text{sumC2} = \left(\sum_{i=0}^{n} \text{c2\_blocks[i]} \right) \mod p$$

Everything placed in a single equation to find sigma of the second message:

$$\sigma_2 = (((\text{tag2}) - v \cdot \text{sumC2}) \cdot \text{mod\_inverse}(1 - v \cdot n, p)) \mod p$$

We have finally found the sigma for the other message, this means we are ready to decrypt.

$$\text{plaintext} = \bigoplus_{c2_{block} \in c2\_blocks} (((c2_{block}[i]) - \sigma_2) \mod p)$$

The result of cracking the ciphered message:

```
b'Congrats! The secret is cozening'
```

3 HMac

3.1 Introduction

The problem with this implementation is that it allows for the forgery of a valid MAC using a key whose value we do not know. Knowing the MAC of the last block of the message, we can forge a new message by simply appending an additional block to the original message. This flaw undermines the integrity and security of the message authentication code (MAC) system.

3.2 The Exploit

To exploit this vulnerability, we begin by padding the last block of the original message to meet the required block size. Next, we append any value to this padded block and pad again to ensure the new block is of the correct size. We then calculate the MAC for this newly modified message by leveraging the tag from the initial message. Specifically, we use AES to encrypt the original tag, using the newly generated block as the key. The output of this encryption is XORed with the original tag to produce a new tag. This method allows us to forge the MAC without knowing the actual encryption key, effectively bypassing the security measures.

m is our original message that we want to forge.

previous_mac is the tag that was generated with the original message.

new_amount is the amount we want to add to our transaction, in this case this is the value we want to forge into our message.

```
def verify(message, key, tag):
    return mac(message, key) == tag

def create_new_message(m, previous_mac, new_amount):
    m = pad(m) # pad the last block
    m += new_amount # add amount to create a new block
    mPrime = m # retrieve forged message
    m = pad(m) # pad the block to calculate the new tag
    blocks = [m[i:i + 16] for i in range(0, len(m), 16)] # transform into blocks
    # calculate the new mac for the last new block that has been added
    h = previous_mac
    h = strxor(AES.new(blocks[-1], AES.MODE_ECB).encrypt(h), h)
    return h, mPrime
```

So now we have **mPrime** that will be able to validate the verify function with our new tag without knowing the value of the **key**.

```
verify(m, k, mc) # return = true, m = original message, k = original key, mc = tag
verify(mPrime, k, newMac) # return = true, mPrime = our modified message, k = still the original key, newMac = says for itself
```

This is an issue because knowing that our intended message was:

```
'Sender: Alexandre Duc; Destination account 12-1234-12. Amount CHF123'
```

We were able to make a new message though it's still passing the verification, turning to this:

```
'Sender: Alexandre Duc; Destination account 12-1234-12. Amount CHF123           800'
```